Crate commonware_cryptography Copy item path

Settings

Help

Summary

Source

Generate keys, sign arbitrary messages, and deterministically verify signatures.

Status

`commonware-cryptography` is ALPHA software and is not yet recommended for production use. Developers should expect breaking changes and occasional instability.

Re-exports

- `pub use bls12381::Bls12381;`
- `pub use ed25519::Ed25519;`
- `pub use sha256::Sha256;`

Modules

- bls12381
- Distributed Key Generation (DKG), Resharing, Signatures, and Threshold Signatures over the BLS12-381 curve.
- ed25519
- Ed25519 implementation of the `Scheme` trait.
- sha256
- SHA-256 implementation of the `Hasher` trait.

Traits

§

- Hasher

- Interface that commonware crates rely on for hashing.
- Scheme
- Interface that commonware crates rely on for most cryptographic operations.

Type Aliases

- Digest
- Byte array representing a hash digest.
- PrivateKey
- Byte array representing an arbitrary private key.
- PublicKey
- Byte array representing an arbitrary public key.
- Signature
- Byte array representing an arbitrary signature.

commonware_cryptography

Module bls12381 Copy item path

Settings

Help

Summary

Source

Distributed Key Generation (DKG), Resharing, Signatures, and Threshold Signatures over the BLS12-381 curve.

Features

This crate has the following features:

- `portable`: Enables `portable` feature on `blst` (https://github.com/supranational/blst?tab=readme-ov-file#platform-and-language-compatibility).

Benchmarks

The following benchmarks were collected on 12/14/24 on a MacBook Pro (M3 Pro, Nov 2023).

```
cargo bench
```

DKG Recovery (Contributor)

```
n=5 t=2      time:   [2.7610 µs 2.8205 µs 2.8763 µs]
n=10 t=4     time:   [9.0694 µs 9.1342 µs 9.1727 µs]
n=20 t=7     time:   [26.200 µs 26.328 µs 26.516 µs]
n=50 t=17    time:   [138.97 µs 139.60 µs 140.10 µs]
n=100 t=34   time:   [538.38 µs 539.65 µs 541.33 µs]
n=250 t=84   time:   [3.4278 ms 3.4504 ms 3.4724 ms]
  n=500 t=167 time:   [13.316 ms 13.387 ms 13.465 ms]
```

DKG Reshare Recovery (Contributor)

```
conc=1 n=5 t=2          time:   [149.16 µs 149.59 µs 149.93 µs]
conc=2 n=5 t=2          time:   [98.315 µs 100.03 µs 101.60 µs]
conc=4 n=5 t=2          time:   [123.41 µs 124.70 µs 126.02 µs]
conc=8 n=5 t=2          time:   [171.07 µs 171.62 µs 172.38 µs]
conc=1 n=10 t=4         time:   [513.59 µs 514.92 µs 516.07 µs]
conc=2 n=10 t=4         time:   [289.33 µs 289.80 µs 290.16 µs]
conc=4 n=10 t=4         time:   [194.84 µs 204.51 µs 217.60 µs]
conc=8 n=10 t=4         time:   [248.80 µs 254.10 µs 256.54 µs]
conc=1 n=20 t=7         time:   [1.3599 ms 1.3609 ms 1.3620 ms]
conc=2 n=20 t=7         time:   [812.24 µs 815.59 µs 818.17 µs]
conc=4 n=20 t=7         time:   [467.86 µs 468.96 µs 470.35 µs]
conc=8 n=20 t=7         time:   [494.17 µs 496.83 µs 499.32 µs]
conc=1 n=50 t=17        time:   [8.9180 ms 9.2827 ms 9.7243 ms]
conc=2 n=50 t=17        time:   [4.8345 ms 4.8378 ms 4.8404 ms]
conc=4 n=50 t=17        time:   [2.8024 ms 2.8074 ms 2.8108 ms]
conc=8 n=50 t=17        time:   [1.8140 ms 1.8219 ms 1.8301 ms]
conc=1 n=100 t=34       time:   [39.960 ms 40.785 ms 42.373 ms]
conc=2 n=100 t=34       time:   [20.347 ms 20.358 ms 20.371 ms]
conc=4 n=100 t=34       time:   [11.211 ms 11.276 ms 11.354 ms]
conc=8 n=100 t=34       time:   [6.9775 ms 7.4456 ms 8.3087 ms]
conc=1 n=250 t=84       time:   [290.29 ms 291.91 ms 295.00 ms]
conc=2 n=250 t=84       time:   [148.39 ms 148.51 ms 148.64 ms]
conc=4 n=250 t=84       time:   [77.728 ms 78.242 ms 78.747 ms]
conc=8 n=250 t=84       time:   [47.123 ms 47.732 ms 48.640 ms]
conc=1 n=500 t=167      time:   [1.4806 s 1.5021 s 1.5357 s]
conc=2 n=500 t=167      time:   [747.69 ms 759.86 ms 779.92 ms]
conc=4 n=500 t=167      time:   [392.03 ms 393.94 ms 395.93 ms]
  conc=8 n=500 t=167      time:   [232.44 ms 238.31 ms 247.56
  ms]
```

Partial Signature Aggregation

```
n=5 t=3                 time:   [126.85 µs 128.50 µs 130.67
µs]
n=10 t=7                time:   [378.70 µs 386.74 µs 397.13
µs]
n=20 t=13               time:   [764.59 µs 777.71 µs 796.76
µs]
n=50 t=33               time:   [2.1320 ms 2.1399 ms 2.1547
ms]
n=100 t=67              time:   [5.0113 ms 5.0155 ms 5.0203
ms]
n=250 t=167             time:   [16.922 ms 16.929 ms 16.937
ms]
  n=500 t=333             time:   [37.642 ms 37.676 ms 37.729
  ms]
```

Signature Generation (Signing)

```
ns_len=9 msg_len=5      time:   [232.12 µs 233.63 µs 235.42
µs]
```

Signature Verification

```
ns_len=9 msg_len=5      time:   [980.92 µs 981.37 µs 981.88
µs]
```

Signature Aggregation (Same Public Key)

```
msgs=10                 time:   [11.731 µs 12.516 µs 13.316
µs]
msgs=100                time:   [117.02 µs 117.16 µs 117.37
µs]
msgs=1000               time:   [1.1751 ms 1.1777 ms 1.1803
ms]
  msgs=10000              time:   [11.878 ms 11.966 ms 12.068
  ms]
```

Aggregate Signature Verification (Same Public Key)

```
conc=1 msgs=10          time:   [1.9960 ms 2.0150 ms 2.0263
ms]
conc=2 msgs=10          time:   [1.3962 ms 1.3979 ms 1.3998
ms]
```

```
conc=4 msgs=10          time:   [1.1857 ms 1.1882 ms 1.1906
ms]
conc=8 msgs=10          time:   [1.1787 ms 1.1873 ms 1.2022
ms]
conc=16 msgs=10         time:   [1.3770 ms 1.3882 ms 1.4133
ms]
conc=1 msgs=100         time:   [12.687 ms 12.704 ms 12.723
ms]
conc=2 msgs=100         time:   [6.8790 ms 6.9518 ms 7.0950
ms]
conc=4 msgs=100         time:   [3.9784 ms 3.9912 ms 4.0085
ms]
conc=8 msgs=100         time:   [2.8804 ms 2.9236 ms 2.9558
ms]
conc=16 msgs=100        time:   [2.7870 ms 2.8007 ms 2.8139
ms]
conc=1 msgs=1000        time:   [119.06 ms 119.11 ms 119.17
ms]
conc=2 msgs=1000        time:   [61.170 ms 61.244 ms 61.332
ms]
conc=4 msgs=1000        time:   [31.822 ms 31.882 ms 31.948
ms]
conc=8 msgs=1000        time:   [19.635 ms 19.991 ms 20.547
ms]
conc=16 msgs=1000       time:   [16.950 ms 17.039 ms 17.126
ms]
conc=1 msgs=10000       time:   [1.1826 s 1.1905 s 1.2018 s]
conc=2 msgs=10000       time:   [603.82 ms 610.05 ms 618.48
ms]
conc=4 msgs=10000       time:   [309.44 ms 312.92 ms 318.01
ms]
conc=8 msgs=10000       time:   [187.57 ms 192.75 ms 198.37
ms]
conc=16 msgs=10000      time:   [158.16 ms 161.60 ms 165.44
ms]
conc=1 msgs=50000       time:   [5.9263 s 5.9377 s 5.9547 s]
conc=2 msgs=50000       time:   [3.0152 s 3.0266 s 3.0417 s]
conc=4 msgs=50000       time:   [1.5420 s 1.5458 s 1.5500 s]
conc=8 msgs=50000       time:   [925.32 ms 929.07 ms 933.83
ms]
  conc=16 msgs=50000      time:   [769.73 ms 773.88 ms 777.97
  ms]
```

Modules

- dkg
- Distributed Key Generation (DKG) and Resharing protocol for the BLS12-381 curve.
- primitives
- Operations over the BLS12-381 scalar field.

Structs

- Bls12381
- BLS12-381 implementation of the `Scheme` trait.

Module ed25519 Copy item path

Settings

Help

Summary

Source

Ed25519 implementation of the `Scheme` trait.

This implementation uses the `ed25519-consensus` crate to adhere to a strict set of validation rules for Ed25519 signatures (which is necessary for stability in a consensus context). You can read more about this here.

Example

```
use commonware_cryptography::{Ed25519, Scheme};
use rand::rngs::OsRng;

// Generate a new private key
let mut signer = Ed25519::new(&mut OsRng);

// Create a message to sign
let namespace = b"demo";
let msg = b"hello, world!";

// Sign the message
let signature = signer.sign(namespace, msg);
```

```
// Verify the signature
   assert!(Ed25519::verify(namespace, msg,
   &signer.public_key(), &signature));
```

Structs

- Ed25519
- Ed25519 Signer.

commonware_cryptography

Module sha256 Copy item path

Settings

Help

Summary

Source

SHA-256 implementation of the `Hasher` trait.

Structs

- Sha256
- SHA-256 hasher.

Trait Hasher Copy item path

Settings

Help

Summary

Source

```
pub trait Hasher:
    Clone
    + Send
    + Sync
    + 'static {
```

```
    // Required methods
    fn new() -> Self;

    fn update(&mut self, message: &[u8]);

    fn finalize(&mut self) -> Digest;

    fn reset(&mut self);

    fn validate(digest: &Digest) -> bool;

    fn len() -> usize;

    fn random<R: Rng + CryptoRng>(rng: &mut R) -> Digest;
}
```

Interface that commonware crates rely on for hashing.

Hash functions in commonware primitives are not typically hardcoded to a specific algorithm (e.g. SHA-256) because different hash functions may work better with different cryptographic schemes, may be more efficient to use in STARK/SNARK proofs, or provide different levels of security (with some performance/size penalty).

This trait is required to implement the `Clone` trait because it is often part of a struct that is cloned. In practice, implementations do not actually clone the hasher state but users should not rely on this behavior and call `reset` after cloning.

Required Methods

Source

`fn new() -> Self`

Create a new hasher.

Source

`fn update(&mut self, message: &[u8])`

Append message to previously recorded data.

Source

```
fn finalize(&mut self) -> Digest
```

Hash all recorded data and reset the hasher to the initial state.

Source

```
fn reset(&mut self)
```

Reset the hasher without generating a hash.

This function does not need to be called after `finalize`.

Source

```
fn validate(digest: &Digest) -> bool
```

Validate the digest.

Source

```
fn len() -> usize
```

Size of the digest in bytes.

Source

```
fn random<R: Rng + CryptoRng>(rng: &mut R) -> Digest
```

Generate a random digest.

Warning

This function is typically used for testing and is not recommended for production use.

Dyn Compatibility

This trait is not dyn compatible.

In older versions of Rust, dyn compatibility was called "object safety", so this trait is not object safe.

Implementors

Source

`impl Hasher for Sha256`

commonware_cryptography

Trait Scheme

Copy item path

Settings

Help

Summary

Source

```
pub trait Scheme:
    Clone
    + Send
    + Sync
    + 'static {
    // Required methods
    fn new<R: Rng + CryptoRng>(rng: &mut R) -> Self;

   fn from(private_key: PrivateKey) -> Option<Self>;

   fn from_seed(seed: u64) -> Self;

   fn private_key(&self) -> PrivateKey;
```

```
    fn public_key(&self) -> PublicKey;

    fn validate(public_key: &PublicKey) -> bool;

    fn sign(&mut self, namespace: &[u8], message: &[u8]) ->
Signature;

    fn verify(
        namespace: &[u8],
        message: &[u8],
        public_key: &PublicKey,
        signature: &Signature,
    ) -> bool;

    fn len() -> (usize, usize);
}
```

Interface that commonware crates rely on for most cryptographic operations.

Required Methods

Source

`fn new<R: Rng + CryptoRng>(rng: &mut R) -> Self`

Returns a new instance of the scheme.

Source

`fn from(private_key: PrivateKey) -> Option<Self>`

Returns a new instance of the scheme from a secret key.

Source

```
fn from_seed(seed: u64) -> Self
```

Returns a new instance of the scheme from a provided seed.

Warning

This function is insecure and should only be used for examples and testing.

Source

```
fn private_key(&self) -> PrivateKey
```

Returns the serialized private key of the signer.

Source

```
fn public_key(&self) -> PublicKey
```

Returns the serialized public key of the signer.

Source

```
fn validate(public_key: &PublicKey) -> bool
```

Verify that a public key is well-formatted.

Source

```
fn sign(&mut self, namespace: &[u8], message: &[u8]) ->
Signature
```

Sign the given message.

The message should not be hashed prior to calling this function. If a particular scheme requires a payload to be hashed before it is signed, it will be done internally.

To protect against replay attacks, it is required to provide a namespace to prefix any message. This ensures that a signature meant for one context cannot be used unexpectedly in another (i.e. signing a message on the network layer can't accidentally spend funds on the execution layer).

Source

```
fn verify(
    namespace: &[u8],
    message: &[u8],
    public_key: &PublicKey,
    signature: &Signature,
) -> bool
```

Check that a signature is valid for the given message and public key.

The message should not be hashed prior to calling this function. If a particular scheme requires a payload to be hashed before it is signed, it will be done internally.

Because namespace is prepended to message before signing, the namespace provided here must match the namespace provided during signing.

Source

```
fn len() -> (usize, usize)
```

Returns the size of a public key and signature in bytes.

Dyn Compatibility

This trait is not dyn compatible.

In older versions of Rust, dyn compatibility was called "object safety", so this trait is not object safe.

Implementors

Source

`impl Scheme for Bls12381`

Source

`impl Scheme for Ed25519`